VR game for children with physical disabilities

Maria Broe Marstrand
Nikolaj Kjældgaard

June 5, 2023

Abstract

Children with physical disabilities face many obstacles in their daily lives, such as difficulties in developing motor skills, socializing with peers, and building self-confidence. In this study, we propose the development of a virtual reality (VR) game to assist in building strength, coordination, and confidence in children with physical disabilities aged 10-18. We identify several requirements to ensure that the game effectively assists children in improving their motor skills while being engaging and immersive.

The requirements include the ability for children to stretch their arms and bodies during gameplay, engage in cross-movements, develop eye-hand coordination, avoid frustration and impossible gameplay, train their core and spine, and improve their balance. Additionally, the game must have replay value to maintain the children's interest over time.

We discuss how we plan to meet these requirements through the use of components in Unity, including Rigidbody and Collider components, and the creation of script components to modify functionality and create new components. We also discuss the use of random object generators, point scoring systems, and high-score tracking to promote replayability and friendly competition.

Overall, this study proposes the development of a VR game that can assist in the improvement of motor skills, socialization, and confidence-building in children with physical disabilities. The game has the potential to provide a fun and engaging way for children to improve their abilities while enjoying an immersive and interactive virtual environment.

Contents

1 Introduction

This project focuses on the field of physical computing, with a specific emphasis on utilizing virtual reality (VR) gaming to improve the motor skills of children with physical disabilities. The aim is to create an engaging and immersive VR game that encourages specific movements, while also fostering a sense of inclusivity and building self-confidence among the targeted age group of 10-18 years.

1.1 Background

Through conversations with a physiotherapist specializing in working with children with physical disabilities, we learned about a project that utilized VR gaming as a tool to motivate children to perform the necessary exercises for better motor development. These exercises not only enhance motor skills but also contribute to the development of self-confidence in these children. Additionally, the physiotherapist highlighted the social limitations faced by these children in participating in games that are not accessible to them due to their physical disabilities. Furthermore, it has been noted that the current available game tends to become monotonous and lose its appeal quickly, according to feedback from the children themselves.

1.2 Motivation

Given the insights shared by the physiotherapist, our motivation is to design a VR game that not only encourages children to engage in exercises that benefit their motor skills but also provides them with an inclusive and enjoyable gaming experience. Our vision is to create a space where these children can play alongside their peers, have fun, and build their self-confidence.

1.3 Research Question

The central research question guiding this project is:

"How can a VR game be designed to improve the motor skills of children with physical disabilities aged 10-18 by encouraging specific movements?"

By addressing this research question, we aim to contribute to the development of solutions that utilize VR gaming as a means of enhancing motor skills, promoting self-confidence, and creating an inclusive gaming environment for children with physical disabilities.

2 Requirements

In this section, we will discuss the key requirements that influenced the development of the game. These requirements stem from three primary sources: the input provided by the physiotherapist, the technical specifications, and our own considerations.

The requirements outlined by the physiotherapist are made to meet the unique needs of the children with physical disabilities who will be using the game. These requirements play a vital role in ensuring that the game effectively caters to their therapeutic and developmental needs.

The technical requirements involve the constraints and specifications imposed by the development platform, Unity, as well as the VR equipment provided by Roskilde University. These requirements play a crucial role in guiding the technical implementation of the game, ensuring compatibility, and achieving optimal performance.

In addition to the physiotherapist and technical requirements, our team also established a set of additional requirements. These requirements reflect our own objectives, creative vision, and desired outcomes for the game. They encompass aspects such as gameplay mechanics, user experience, aesthetics, and overall enjoyment for the target audience.

By considering and integrating these requirements, we aim to create a game that meets the specific therapeutic goals, technical standards, and creative aspirations set forth by the various stakeholders involved.

2.1 Requirements from the physiotherapist

In collaboration with a physiotherapist, we have established specific requirements for the motor movements that the player must perform during the game. These requirements are as follows:

2.1.1 Stretch of arms and body

The first requirement is for the children to be able to stretch their arms and bodies during gameplay. To measure the extent to which this is achieved, a pre-game measurement system has been proposed. This will involve recording the maximum upward, downward, and diagonal movement of the child's hands and legs.

2.1.2 Cross movements

The second requirement is for the children to engage in cross-movements during gameplay. To encourage this, the game will feature two distinct colours for each hand of the child, and crates in corresponding colours that they will need to hit with the corresponding hand. To further encourage cross-movements, the crates will be placed in opposite positions, with the left hand required to hit a crate on the right side, for example.

2.1.3 Eye-hand coordination and avoidance of frustration and impossible gameplay

One of the challenges faced by some children is the development of eye-hand coordination. This deficiency can lead to difficulties in engaging in various social activities with peers, such as catching a ball, drawing, or typing on a keyboard. As such, it is critical to develop a game that effectively enhances this ability. To ensure that this requirement is met in our VR game, we will incorporate obstacles that the children must avoid and objects that they must grip or hit with either their left or right hand.

These objects will be specific to the hand required for the task, and clear cues will be provided to indicate the appropriate hand to use. Additionally, the game will start with a slow and easy pace, gradually increasing in difficulty as the child becomes more adept at the gameplay. To prevent frustration and promote confidence-building, a mechanism to decrease the difficulty level will be implemented if the child struggles. This mechanism will not be visible to the child, but will ease the gameplay to encourage persistence and mastery.

2.1.4 Core-, balance-, and spine training

To address the requirement for core-, balance-, and spine training, we will incorporate exercises that encourage correct movements and strength-building. Specifically, we will include obstacles that require the player to squat and maintain an upright posture while looking straight ahead. For instance, we may create crates that can only be hit when the player is looking straight ahead and squatting, and if not hit in time, they will hit the player and be made of an unbreakable material. These exercises will help to promote good posture and core strength in the children.

2.1.5 Variation in gameplay

The game must have replay value to maintain the children's interest over time, as improving their skills requires practice. Children have expressed that the game becomes tedious after playing it for a while. Therefore, it is crucial that the game is engaging and has enough variety to keep them entertained.

To address this issue, we plan to incorporate a somewhat random object generator into the game so that the gameplay is not always the same. Additionally, we will introduce a point scoring system that allows children to earn points for upgrading their character or unlocking new levels. It is also important that the children can see their high-scores to encourage friendly competition between themselves and their friends.

2.2 The technical requirements

When developing a VR game and using borrowed equipment from Roskilde University, it is important to meet certain technological requirements. These requirements are as follows:

2.2.1 Run on Quest 1

As we have borrowed equipment from Roskilde University, it is crucial that our game remains compatible with the Oculus Quest Pro 1. The Oculus Quest Pro 1 is a virtual reality headset that was released some time ago, and it may lack some of the advanced features and capabilities found in more recent VR devices.

Considering the age of the Oculus Quest Pro 1, it may not support the latest hardware advancements or offer the same level of performance as newer VR headsets. Its processing power, display resolution, and tracking capabilities might be limited compared to the latest models. Additionally, the software development tools and frameworks available for the Oculus Quest Pro 1 may not incorporate the most recent updates and optimizations found in modern game development platforms like Unity.

Navigating these challenges requires careful consideration of the device's limitations, optimizing the game's performance to ensure smooth gameplay, and exploring creative solutions to deliver an immersive experience despite the older technology.

2.2.2 Game engine / Unity / C#

For our project, we have selected Unity as the game engine to develop our VR 3D game specifically designed for the Oculus Quest 1. Unity offers a comprehensive and user-friendly platform that combines powerful features and intuitive design, making it the perfect choice for our project.

A notable advantage of Unity is its built-in support for the C# programming language. With C# as our programming language of choice, we can implement game logic and scripting seamlessly. And by using C# in Unity, we can optimize performance to deliver a smooth and captivating experience on the Oculus Quest 1.

With Unity as our game engine and the flexibility of C# programming, we are confident in our ability to create an immersive VR game that can maximize the capabilities of the Oculus Quest 1.

2.2.3 VR compatible / XR toolkit

TEXT

2.3 Additional requirements

In addition to the requirements set by the physiotherapist and the technical specifications, we have also implemented our own set of requirements that we strive to fulfill. These requirements are as follows:

2.3.1 Clean code

Clean code is a high priority in our project. We aim to write code that is easy to understand and maintain, allowing us and other developers to extend and improve the game effortlessly. By prioritizing clean code, we want to ensure a smoother development process and pave the way for future enhancements. By clean code we mean:

- **Readable:** Clear naming, proper indentation, and organized structure for easy understanding.
- **Maintainable:** Designed to facilitate updates, modifications, and extensions without introducing errors.
- **Elegant:** Achieving simplicity, conciseness, and clarity while avoiding unnecessary complexity.
- **Self-explanatory:** Clearly expressing its purpose and functionality without excessive comments.
- **Modular:** Divided into smaller, reusable modules that encapsulate specific tasks or responsibilities.
- **Efficient:** Optimized for performance and resource usage, avoiding unnecessary computations.
- **Scalable:** Designed to easily accommodate future growth and adapt to changing requirements.

- **Testable:** Structured to enable comprehensive testing, ensuring correctness and reliability.

3 Concept clarification

As we are using Unity Engine to develop our game, we want to establish a clear understanding of some of the terms that we will be referring to throughout the report. These terms relate to the functionality and features of the Unity Engine and how we intend to incorporate these elements into our game.

3.1 Unity

Unity is a popular game engine that offers a range of tools and features for game development. It provides a powerful and user-friendly environment that enables the creation of interactive and immersive experiences. Key components of Unity include the scene, which represents the visual representation of the game state, the hierarchy for organizing game objects, and the inspector for modifying object properties and settings. Please also see 2.2.2 for more.

3.1.1 Scene

In Unity, a scene refers to the visual representation of the game state. It has various elements and assets that make up a specific level or gameplay area. Scenes serve as the canvas where game objects and their interactions are designed and orchestrated.

3.1.2 Hierarchy

The hierarchy in Unity is a structure that organizes instances of game objects, including both prefabs (reusable object templates) and non-prefabs. It represents the parent-child relationships between game objects, allowing for hierarchical organization and management of the game's elements.

3.1.3 Inspector

The inspector is a crucial component of Unity's interface. It provides a detailed view of the selected game object's components, properties, and settings. Through the inspector, we can modify and fine-tune the behaviour and appearance of game objects by adjusting their associated components and parameters.

3.2 Components

Components in Unity are modular building blocks that can be combined in various ways to create specific functionalities and behaviours for game objects. This flexibility allows us, as game developers, to create complex game objects that can be picked up and manipulated by the player, improving the player's experience in the virtual environment [1].

For example, in our VR game project, we will use the predefined Rigidbody and Collider components in Unity to create game objects that behave realistically and can interact with the player and other game objects. Additionally, we will create a script component to modify the functionality of our game objects and to create new components to meet our specific needs.

To view the list of components attached to a GameObject in the Inspector window, you should select a GameObject either in the Hierarchy window or in the Scene view. It is possible to attach multiple components to a GameObject; however, every GameObject must possess one and only one Transform component. The reason being that the Transform component dictates the GameObject's location, rotation, and scale.

Overall, the use of components in Unity allows us to create unique and interactive virtual environments that improves the player's experience. The modularity of components also makes it easy to reuse them in other game objects.

3.2.1 Rigidbody

We will use the 'Rigidbody' component to control the position of objects through physics simulation. By assigning a Rigidbody component to a game object, we enable Unity's physics engine to take charge of its movement, allowing it to respond realistically to physical forces, such as gravity, and interact seamlessly with other objects, provided that a compatible Collider component has been added. With these components in place, the object's behavior follows the laws of physics, creating a more immersive and captivating gaming experience [2].

3.2.2 Transform

The Transform component in Unity is a fundamental building block for defining the crucial properties of a game object. It encapsulates the object's position, rotation, and scale, empowering precise control over its placement, orientation, and size within the 3D environment.

With the Transform component, you can not only manipulate and animate objects but also establish hierarchical structures, perform calculations related to positioning and orientation, and enable interactions with other elements in the game [3].

Additionally, in our project, we will utilize the Transform component's ability to adjust the position property for object movement. Specifically, we will employ this feature to move interactive objects towards the player, enhancing the overall experience.

3.2.3 Colliders

Colliders play a crucial role in defining the physical boundaries and interaction properties of the interactive objects in our game, allowing for realistic collisions and interactions between them [4]. Unity offers a diverse range of collider options. The list of colliders that we will make use of:

- Box Collider: This shape creates a collider with a rectangular box, ideal for objects with well-defined edges and corners.
- Sphere Collider: Using this shape, we can create a collider that encompasses objects within a spherical boundary, providing a realistic representation for round or curved objects.
- Mesh Collider: By utilizing a custom mesh, we have the flexibility to define colliders that accurately match intricate geometries. This allows for precise collision detection and interaction with objects of any shape.

Incorporating these colliders into our game ensures that objects possess accurate physical boundaries, enabling immersive experiences with lifelike collision and interaction mechanics in our Unity VR 3D environment.

3.2.4 Materials

Materials refer to components that define the visual properties of objects in a 3D scene. They control aspects such as color, texture, reflectivity, transparency, and other visual characteristics.

Materials in Unity consist of various properties, including the shader, which determines how light interacts with the object's surface, the albedo property that defines the base color or texture, while metallic represents the object's metalness and smoothness controls the surface roughness or glossiness, and normal maps simulate fine details [5].

By adjusting these material properties, we can create a wide range of visual effects to the objects utilized within the game.

3.2.5 Scripts

Scripts are a fundamental component used to define the behaviour and functionality of objects within a game or application.

They play a crucial role in creating interactive and dynamic experiences by allowing us, as developers, to write custom code that controls the behaviour of game objects, implements game mechanics, and handles user interactions.

Scripts in Unity are attached to game objects as components. Each script contains a collection of functions, variables, and logic that dictate how the associated object should behave and respond to different events or inputs. These scripts, therefore, make it possible to create an actual game where users can interact with the objects and we, as developers, can create an environment worth playing in. Note that the scripts we are using, are written in C#.

3.3 Objects and Prefabs

In Unity, game objects are the basic building blocks of a scene. Game objects can have various components attached to them, such as the ones we described earlier, to add functionality and behaviours to the object.

Objects can be created in Unity by selecting GameObject from the main menu, or by right-clicking in the Hierarchy panel and selecting Create Empty.

A prefab, on the other hand, is a special type of asset in Unity that allows us to create a template of a game object, that can be reused multiple times throughout the game. A prefab is essentially a pre-configured game object that can be saved as a separate file and reused in different scenes and projects [6].

It is possible to include Prefab instances within other Prefabs, which is known as nesting Prefabs. When Prefabs are nested, they maintain their connections to their own Prefab Assets while simultaneously becoming part of another Prefab Asset. We will make use of this ability, when creating the scene of the game [7].

The main difference between objects and prefabs is that objects are unique instances in a scene, while prefabs are templates that can be reused multiple times in different scenes and projects. However, when a prefab is added to a scene, it creates an instance of that prefab, which is essentially a copy of the original prefab with its own unique identity in the scene. The instance can then be modified as needed, such as changing its position or adding new components, without affecting the original prefab.

3.4 Assets

While we have the capability to create our own assets, we have also taken advantage of the opportunities provided by the Unity Asset Store, that provides a vast collection of ready-made assets, tools, and plugins that developers can leverage to accelerate their projects. It serves as a hub for a wide range of assets, including 3D models, textures, audio files, scripts, and more, created by both Unity Technologies and third-party developers [8].

In our game, assets, particularly prefabs, play a crucial role in shaping our virtual world as prefabs allow us to efficiently create consistent instances of game elements with predefined properties and behaviors. By utilizing assets, we streamline our workflow, enhance visual consistency, and create engaging gameplay experiences.

4 Design and Analysis

In this section, we will show a few approaches to designing different aspects of the game, that we see fit or available. We will then give a short description of each approach and their respective advantages and disadvantages. Finally, we will give a sub-conclusion about our chosen approach and the reasons behind our decision.

4.1 Audio

Sound design plays a crucial role in game development, greatly impacting the player's immersion and overall experience. Although players may not always consciously think about it, the absence or incorrect implementation of sound can significantly diminish the game's quality. Therefore, it is essential to carefully consider and discuss sound design when developing our VR game.

We have examined several options:

4.1.1 Approach 1: No music or sound

This is the simplest approach, but it poses the risk of breaking immersion. In the real world, our senses of touch, sight, and hearing collectively contribute to the immersive experience. Without sound, the game may feel incomplete and less engaging for players.

Advantages:

- Simplicity: Implementing no sound or music requires minimal effort and resources, making it the easiest option to execute.
- Unobtrusive: The absence of sound may allow players to focus solely on the visual aspects of the game, which can be desirable for certain genres or gameplay styles.

Disadvantages:

- Lack of immersion: Sound plays a vital role in creating an immersive experience. Without it, the game may feel less realistic and engaging for players.
- Reduced emotional impact: Sound effects and music can evoke emotions and enhance storytelling, which is lost when no audio is present.
- Monotony: The absence of sound can lead to a monotonous gameplay experience, potentially making the game less enjoyable over time.

4.1.2 Approach 2: Rhythm-based (hits on the beats)

A rhythm-based approach can enhance the game's appeal, but it is considerably more complex to execute. This approach necessitates careful consideration in selecting appropriate audio and effectively coordinating it with in-game objects and actions. Achieving synchronization and maintaining consistency throughout the game can be time-consuming and demanding.

Advantages:

- Enhanced engagement: Incorporating rhythm-based audio can make gameplay more interactive and engaging, as players synchronize their actions with the beat.
- Immersive and synchronized: When executed successfully, this approach can provide a highly immersive experience, where audio and player actions feel interconnected.

Disadvantages:

- Complexity: Finding suitable audio and synchronizing it with gameplay elements can be challenging and time-consuming, requiring careful attention to detail.
- Limited flexibility: Rhythm-based sound design may not align well with our game genre, as we are developing a game specifically designed for children with physical disabilities.
- Skill requirement: Players may need a certain level of rhythm or coordination to fully enjoy and take advantage of the rhythm-based gameplay mechanics.

4.1.3 Approach 3: Sound when hitting an object and music in the background

This approach aims to strike a balance between immersion and ease of implementation. Given that our game caters to children with physical disabilities, it is crucial to provide a gaming experience that focuses on delivering the right sensory stimuli rather than emphasizing synchronized movements with music. This approach allows us to create an immersive environment by incorporating sound effects when objects are interacted with while maintaining a background score that enhances the overall

atmosphere. By prioritizing fun and immersion, we want to create an enjoyable gaming experience for our target audience.

Advantages:

- Accessibility: This approach caters to children with physical disabilities, allowing them to interact with the game and experience feedback through sound when objects are hit.
- Immersion and atmosphere: Incorporating sound effects and background music enhances the overall immersion and creates a more captivating gaming environment.
- Easier implementation: Compared to the rhythm-based approach, this option requires less complexity and effort, making it more accessible for development.

Disadvantages:

- Potential distraction: In some cases, background music may divert players' attention away from critical gameplay elements or instructions.
- Limited synchronization: While sound effects can be triggered by hitting objects, they may not always be perfectly synchronized with the player's actions, which could impact the overall sense of immersion.

4.1.4 Sub-conclusion: Sound when hitting an object and music in the background

After careful consideration of the advantages and disadvantages of each sound design approach, we have chosen Approach 3: Sound when hitting an object and music in the background. This decision aligns with our goal of creating a game that caters to children with physical disabilities while still providing a fun and immersive experience. By incorporating sound effects when objects are interacted with and maintaining a background music track, we can enhance the overall immersion and atmosphere of the game. This approach ensures accessibility for our target audience, allowing them to engage with the game through auditory feedback. Additionally, the ease of implementation associated with this approach allows us to focus on delivering the right sensory stimuli without the added complexity and skill requirements of a rhythm-based approach.

4.2 Movement of the player

At the core of our project, we draw inspiration from VR games where players engage with moving objects through hitting, punching, or other interactions. However, we also hold a deep appreciation for classic 2D games and seek to infuse our concept with a twist.

Our goal is to make the player feel as though they are actively moving towards the interactive objects, rather than just standing still. We aim to create an environment where the player can experience varying speeds, allowing them to move faster or slower. With this in mind, we have explored different options to achieve this illusion.

4.2.1 Approach 1: Avatar-based Walking

One approach is to create an avatar that appears to walk around a virtual environment. This would be visually appealing and immersive, especially since the player is standing up during gameplay. However, considering safety concerns, we must ensure that players do not feel the urge or temptation to physically move during the game. If we were to design an avatar that walks or runs, players might be inclined to mimic those movements in real life. This would compromise safety and potentially break the immersion if the player moved away from the intended scene.

Advantages:

- Visual appeal and immersion due to the player standing up during gameplay.

- Provides a sense of actively moving towards interactive objects.
- Can offer a wide range of movement possibilities within the virtual environment.

Disadvantages:

- Safety concerns as players might be tempted to physically move, potentially leading to accidents or distractions.
- Risk of players moving away from the intended scene, breaking immersion.
- Requires careful design to ensure player safety and prevent unintended consequences.

4.2.2 Approach 2: Moving Vehicle (Mine-Cart)

An alternative approach involves placing the player on a moving vehicle, which would naturally discourage physical movement. Drawing inspiration from older games, we thought a mine-cart would be a fun way to achieve this. By adopting this approach, we can control the speed and create a sense of immersion while still allowing for adjustments in velocity, be it faster or slower.

Advantages:

- Discourages physical movement by placing the player on a moving platform.
- Provides a natural constraint, making it less likely for players to wander or leave the intended play area.
- Allows for controlled speed adjustments, enhancing immersion and gameplay dynamics.

Disadvantages:

- Requires additional design considerations to ensure a smooth and enjoyable experience within the confines of the vehicle's movement.

4.2.3 Sub-conclusion: Moving Vehicle (Mine-Cart)

We have decided to choose Approach 2, which involves placing the player on a moving vehicle, specifically a mine-cart. This approach effectively addresses safety concerns by discouraging physical movement while providing an immersive experience.

The mine-cart concept offers a fun and engaging element to the experience. By placing the player on a moving platform, we naturally discourage wandering or leaving the intended play area, ensuring better control and focus during gameplay.

Implementing this approach requires careful consideration of the design elements within the mine-cart's movement. Smooth navigation, strategically placed obstacles, and interactive elements play a crucial role in ensuring an enjoyable and immersive gameplay experience.

In conclusion, Approach 2, involving the use of a moving mine-cart, provides an effective solution to address safety concerns and promote engagement. With controlled speed and a focus on design considerations, this approach offers an exciting and immersive gameplay experience for our users.

4.3 The players environment

In virtual reality (VR) games, the environment is essential because it helps players feel more immersed in the experience. It sets up the scene and provides helpful cues to guide gameplay. We aimed to create an environment filled with non-interactable elements that simply provide key visual hints to the player, such as trees, rocks, and mountains. Without these environmental features, players wouldn't be able to perceive a sense of movement, since movement is perceived through the positional relation to other objects. Therefore, the main purpose of the environment is to subtly indicate the speed and movement of the game. By having objects like trees and rocks pass by at a certain pace in the game, we can give players a feeling of moving at a certain speed.

4.3.1 Approach 1: Moving The players position through a big environment

Simulating movement by having the player traverse a vast environment is one strategy. This approach necessitates the design of a large map and a mechanism to guide the player through the different zones. Unity's built-in terrain tool, which enables the crafting of intricate terrains, can be particularly useful here. We succeeded in constructing several expansive, highly-detailed terrains, albeit at the cost of significant computational resources for rendering.

A critical factor to consider is the player's movement through the environment. While it may seem intuitive to guide the player through all distinct areas of the vast environment, this method could complicate the implementation of game logic. Our decision to limit player mobility by situating them in a minecart required us to take this into account during the environment design phase.

If we propel the player through the environment via the minecart, we must be able to calculate the location of interactive objects. This could be achieved by manually predefining potential object spawn points, or even spawning these objects based on the minecart's trajectory. The concept of object spawning, however, becomes more complex under these conditions.

Another potential drawback of this approach is the risk of inducing motion sickness. To navigate the track, the minecart would likely need to rotate, which might disorient some players. Balancing immersive gameplay with player comfort remains a central challenge in the design of the environment.

Advantages:

- Traversing a vast environment can be highly immersive for players, offering them a sense of exploration.
- A vast environment offers less repetitive, in terms of what the player observes.
- The utilization of Unity's built-in terrain tool can simplify the process of designing detailed environments.

Disadvantages:

- Building several expansive and highly detailed terrains can consume significant computational resources, which can result in performance issues or high hardware requirements.
- Guiding players through distinct areas can complicate the implementation of game logic.
- The movement of the minecart, particularly rotations, could induce motion sickness in some players, compromising their gaming experience.

4.3.2 Approach 2: Moving a small environment towards the player

At first glance, it might seem counterintuitive to move the environment towards the player rather than moving the player towards it. However, this method has its own merits, primarily its simplicity. Given that the player remains relatively stationary in the minecart, we can take advantage of this characteristic.

This approach opens up the possibility for us to create a compact, richly detailed environment. By moving this environment, we can simulate the impression of player motion, all while preventing the loading of an excess of superfluous objects requiring rendering.

This approach aligns well with the Oculus headset, as we can significantly reduce the size of the environment to focus on a smaller, detailed scenery. One challenge, however, is that the environment will likely comprise numerous elements, each needing to be moved independently. Depending on the number of objects, this could demand substantial computational resources.

A significant challenge lies in preserving the illusion of a vast environment when the player is, in reality, confined to a smaller space. One potential solution involves placing a limit on the player's field of vision without disrupting the immersive qualities of the gameplay. Implementing a form of visual fog could be a practical strategy to achieve this effect, subtly concealing the environment's true boundaries while preserving the player's sense of vastness.

Nevertheless, by bringing the environment towards the player, we can accurately predict where to spawn objects since the player isn't moving and the logic determining where to spawn objects will not require complex calculations based on direction, speed and other factors. This approach results in making logic for object spawning relatively straightforward to implement.

Advantages:

- The method of moving the environment towards the player simplifies the concept of movement, as the player remains relatively stationary, making it easier to manage game mechanics.
- Preventing the loading of excess objects for rendering can enhance the game's performance and reduce the computational resources needed.
- This method aligns well with VR headsets such as the Oculus, which generally work better with smaller, highly detailed environments rather than vast spaces.
- Since the player is stationary, predicting where to spawn objects becomes straightforward, eliminating complex calculations based on player direction, speed, etc.

Disadvantages:

- The environment will likely comprise numerous elements, each needing to be moved independently. Depending on the number of objects, this could demand substantial computational resources.
- Preserving the illusion of a vast environment while the player is confined to a smaller space can be challenging. It might require additional techniques, like visual fog, which can add to the complexity of the game design.
- Placing a limit on the player's field of vision might disrupt the immersive quality of the gameplay, even if this is done subtly.
- Depending on how the moving environment is implemented, players might perceive the environment as repetitive or artificial if they notice patterns in the moving scenery.

4.3.3 Sub-conclusion: Moving a small environment towards the player

After carefully evaluating the two approaches, we have decided to implement the approach where the environment moves towards the player for multiple reasons.

Firstly, this approach offers simplicity in terms of the overall gameplay mechanics. As the players position remains completely stationary, managing the game's dynamics becomes easier, allowing us to concentrate more on enhancing the user experience and less on handling the complications of player movement.

The decision to create a compact, detailed environment also holds significant appeal. This choice aligns well with our aim to provide an immersive and visually engaging experience, all while minimizing the performance cost associated with rendering an expansive environment. The ability to prevent the loading of excess, inessential objects is a valuable advantage in terms of optimizing our game's efficiency and performance.

This approach also offers considerable advantages in terms of its compatibility with VR systems like the Oculus Quest Pro headset.

Another compelling reason is the streamlined logic for object spawning. With the player's position constant, we can more easily predict where to spawn objects, resulting in a simpler and more efficient game design process.

While challenges do exist, such as preserving the illusion of a vast environment, we are confident that creative solutions, like implementing visual fog, can effectively address these issues. The appeal of maintaining a simpler game logic, a highly-detailed environment, and better performance optimization ultimately sways our decision towards moving the environment towards the player. We believe this approach will deliver an engaging, immersive, and seamless gaming experience for our users.

4.4 Correct motoric movements (the concept of the player)

Given that the motivation behind this project is to assist children with physical disabilities, it is vital that they perform the correct movements. To find inspiration, we explored two distinct games with a similar gameplay philosophy where players remain stationary and interact by hitting, punching, or engaging with objects that move toward them.

This exploration provided us with ideas on how we could implement the player's movements. We considered the following approaches:

4.4.1 Approach 1: Sword-Slicing Mechanics

Recognizing the satisfaction of slicing objects, we considered creating a gameplay mechanic where players would hold swords in their hands and slice through crates or other obstacles approaching them. This approach aimed to adapt the game to accommodate any player by making the swords long, ensuring a proper fit for individuals of all sizes.

However, upon testing this approach, we realized that the movements primarily involved the wrists rather than the entire body, including the arms and legs.

Advantages:

- Flexibility and adaptability: The adjustable swords allow customization to suit the player's physical abilities, making the game accessible to a wide range of children with physical disabilities.
- Immersive experience: Holding swords and slicing through objects can create a more engaging and immersive gameplay experience, potentially increasing enjoyment and motivation.

Disadvantages:

- Limited body movement: The primary drawback is that the movements primarily involve the wrists rather than engaging the entire body, including the arms and legs. This restricts the level of physical activity and may not provide the desired therapeutic benefits.

4.4.2 Approach 2: Fist-Based Interaction

During our exploration, we also experimented with a game that involved using boxing gloves. In this game, players would engage their entire bodies, including their legs and arms, and experience physical exertion. However, considering the nature of our chosen setting, being on a moving cart in nature, incorporating boxing gloves felt out of place. Instead, we have decided to focus on hand movements, particularly using a closed fist or even raising a single finger, allowing for added diversity in gameplay.

Advantages:

- Ensures full-body engagement: By emphasizing hand movements, including the use of a closed fist or pointing finger, we encourage players to involve their entire bodies rather than relying solely on wrist motions.
- Increased accessibility for players with different physical disabilities, as hand-based interactions eliminate the need for specific wrist movements, ensuring inclusivity in the gaming experience
- Supports physiotherapy requirements: Approach 2 ensures that the children perform the correct movements as required by their physiotherapist, providing them with beneficial therapeutic exercises.

Disadvantages:

- Lack of a quick-fix solution: Unlike the first approach, implementing fist-based interaction requires the addition of a calibrator to calculate the appropriate distance at which obstacles should be placed relative to the player.

4.4.3 Sub-conclusion: Fist-Based Interaction

After evaluating the advantages and disadvantages of the two approaches for correct motoric movements in our game, we have chosen the preferred method, Approach 2.

Approach 1, Sword-Slicing Mechanics, initially appeared promising with its flexibility and immersive experience. However, it primarily relied on wrist movements, limiting full-body engagement and potentially hindering therapeutic benefits.

In contrast, Approach 2, Fist-Based Interaction, focuses on hand movements like using a closed fist or pointing finger, ensuring full-body engagement and supporting physiotherapy requirements. It promotes inclusivity and provides beneficial therapeutic exercises. Although implementing this approach requires a calibrator for obstacle placement, it aligns better with our objective of encouraging correct movements and an enjoyable gameplay experience.

In conclusion, we have selected Approach 2 as the preferred method for the player's motoric movements. By emphasizing hand movements and involving the entire body, we aim to assist children with physical disabilities in performing correct movements.

4.5 Spawning of interactive objects

Since the spawn of interactive objects serves as the foundation of the game, we have explored different possibilities and approaches throughout the development process. This section presents three distinct approaches for the object spawning mechanism, each with its own set of considerations and design choices. We will describe these approaches in detail, outlining the thought process behind each approach, along with their respective pros and cons. Ultimately, we will highlight the final chosen approach.

4.5.1 Approach 1: Random Spawning

In this approach, we consider the implementation of random spawning for crates and obstacles in the game. The concept revolves around creating a "spawner" game object that will generate objects moving towards the player. The objective is to introduce an element of unpredictability and challenge to the gameplay experience by requiring the player to interact with these randomly spawned objects.

To implement random spawning, a random function can be utilized to select spawning positions from a predefined list of available positions. The spawner can then generate crates and obstacles in various colors, such as blue or red, providing a visually diverse experience. The player would be required to interact with these objects by performing actions like punching, touching, or avoiding them based on specific game criteria.

Advantages:

- Adds an element of unpredictability and excitement to the gameplay.
- Enhances replayability by generating different object arrangements in each playthrough.

Disadvantages:

- Difficulties in ensuring that the required movements are performed in a desired order.
- Randomness may occasionally result in less optimal gameplay sequences.

By exploring random spawning as a approach, the game can offer a dynamic and engaging experience for players. However, careful consideration must be given to ensure that the gameplay remains coherent and enjoyable, addressing any potential challenges related to the order of required movements and optimizing the randomness for a balanced gameplay flow.

4.5.2 Approach 2: Series-based Spawning

One approach we considered was series-based spawning, managed with a switch-case, which aimed to introduce a series of objects in a specific order to guide the players' movements. This approach

offered advantages such as consistent spawning order, enhanced replayability, and unique gameplay experiences. However, it also posed challenges in terms of scalability and code complexity. The pros and cons of this approach in more detail:

Advantages:

- Ensures consistent spawning order, facilitating precise movement requirements.
- Adds unpredictability and variation, enhancing replayability.
- Provides a unique and engaging experience for children, promoting motivation.
- Allows for the creation of different series with varied difficulty levels.

Disadvantages:

- May not be scalable for multiple levels or variations.
- Requires careful management of the switch-case/state-machine approach.
- Balancing randomness and predictability can be challenging with a switch-case approach.
- Complexity in designing the series and spawn position system.
- Increased development and testing efforts.
- Potential technical issues and bugs specific to series-based spawning/shuffle method within a switch-case.

While the series-based spawning approach provides benefits such as consistent spawning order and enhanced replayability, it is important to consider its scalability limitations, particularly when incorporating additional levels. The switch-case implementation, while offering order and variation, may become less manageable and more challenging to maintain as the number of levels increases. Careful planning and consideration should be given to ensure that the game architecture remains scalable and flexible.

4.5.3 Approach 3: Incorporating SeriesManager, Level, and LevelManager

One potential approach to consider in the game development process is the incorporation of three important components: SeriesManager, Level, and LevelManager.

The SeriesManager script would serve as a central coordinator, responsible for managing different series of objects in the game. Its role would be to ensure that each object is correctly positioned in the game world and to provide a convenient method for retrieving specific series based on their names.

The Level class would represent individual levels within the game. It would maintain a list of series names that are required to be spawned. At the beginning of each level, the Level class would prepare the series list by creating instances of the corresponding Series class, using the names provided by the SeriesManager. It would then shuffle the series list and initiate the controlled spawning of the series' objects. Once all series have finished spawning their objects, the Level would notify the LevelManager that the current level has been completed.

The LevelManager, implemented as a Singleton, would take charge of the overall progression of levels in the game. It would keep track of the current level and provide methods to start a level and transition to the next level once the current one is successfully completed.

By incorporating these components, several advantages can be gained. Firstly, this approach has the potential to create a more organized and scalable codebase, making future development easier. Secondly, the clear separation of responsibilities among the SeriesManager, LevelManager, and Level classes would grant better control over the sequencing of gameplay. Thirdly, this approach would address previous challenges related to shuffling and unintended loops. Lastly, it improves the ease of maintenance and flexibility, facilitating the smooth integration of future updates.

Advantages:

- Potential for a more organized and scalable codebase, easing future development.
- Clear separation of responsibilities among the SeriesManager, LevelManager, and Level classes, enabling better control over gameplay sequencing.
- Addresses previous challenges related to shuffling and unintended loops.
- Improves ease of maintenance and flexibility, facilitating smooth integration of future updates.

Disadvantages:

- Introduction of increased complexity due to the implementation of multiple scripts.
- Additional development and testing time may be required for the restructured architecture.

Overall, the adoption of this approach presents promising possibilities for enhancing the structure and functionality of the game. It also provides better control, organization, and scalability while effectively addressing past challenges.

However, it should be acknowledged that it also introduces increased complexity and may require additional development and testing efforts.

4.5.4 Sub-conclusion: Incorporating SeriesManager, Level, and LevelManager

After careful consideration, we have chosen Approach 3, Incorporating SeriesManager, Level, and LevelManager, as the optimal method for spawning interactive objects in our game. This approach introduces key components that enhance the game's structure, functionality, and control. The SeriesManager coordinates different object series, the Level class manages individual levels and their spawning order, and the LevelManager oversees the overall level progression. This approach provides a more organized and scalable codebase, improves gameplay sequencing, and addresses previous challenges. While it introduces increased complexity, the benefits outweigh the drawbacks, ensuring an engaging and dynamic gameplay experience.

5 Implementation

In this section, we will explain how we implemented the selected approach in the different areas of our game. We will also showcase the overall class diagram, illustrating the relationships between the different scripts and components.

5.1 Hardware and Software Implementation Details

- Unity Version - Unity Packages (3D template, Android etc.) - Tools (ProBuilder) - Setup - Assets in project from (Unity Asset Store)

5.2 Class Diagram

In this section, we will delve into the detailed analysis of the class diagram for our VR game. The class diagram provides a comprehensive overview of the various components and their relationships within the game architecture. We will begin by presenting the overall view of the class diagram, capturing the essential elements and their interactions. Subsequently, we will explore specific sections of the diagram, focusing on crucial aspects such as the environment, the player, the objects, and the spawning mechanism.

This overall class diagram shown below, presents a holistic representation of the VR game's structure and organization. It illustrates the key classes and their associations, providing insights into the flow and functionality of the game system.

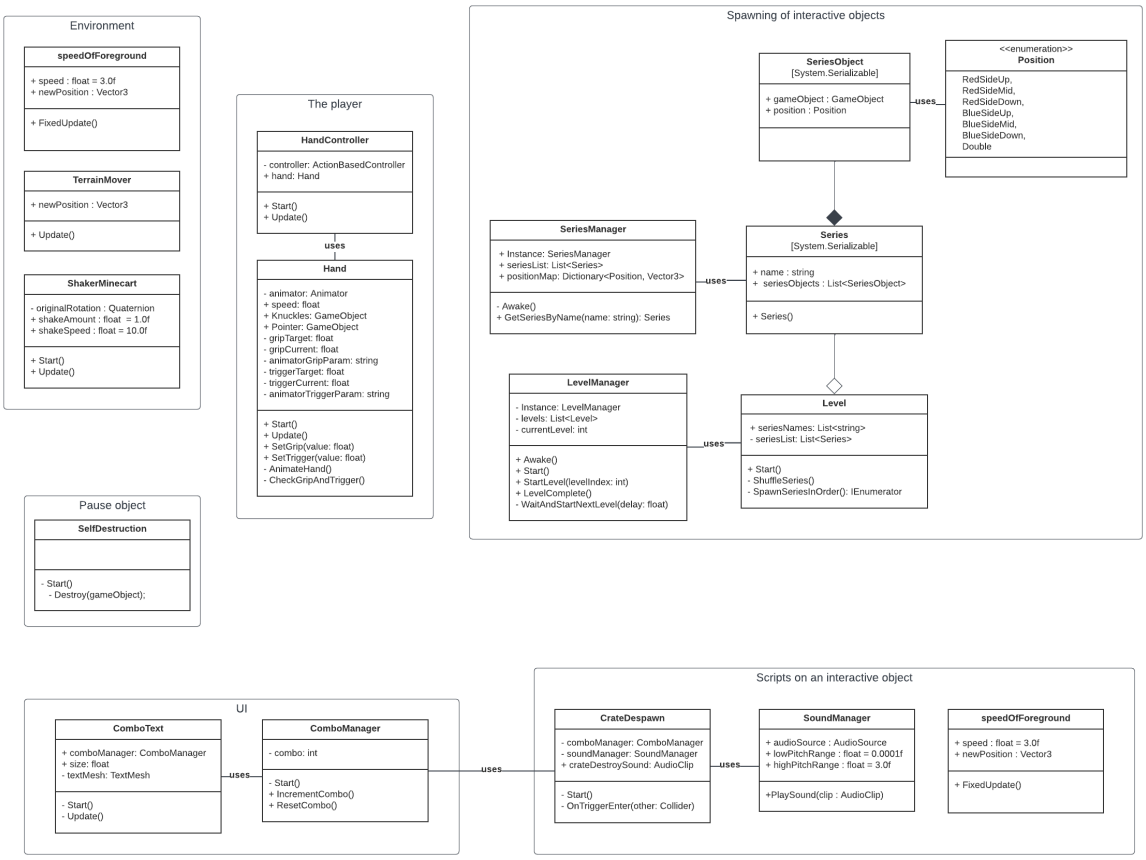

Figure 1: Overview of Class Diagram

5.3 The Player

This section is a description of the concept of the player. It covers the how the player is able to move and view the world in our VR experience.

5.3.1 XR Toolkit and Oculus Integration

In our Unity project, various elements come together to form the player. Utilizing a VR headset, specifically the Oculus Quest Pro, our initial task was to establish a realistic system for movement and vision. We wanted to ensure that our game was compatible with a variety of headsets, leading us to choose the XR Interaction Toolkit. This powerful tool enables our game to function seamlessly across a range of VR platforms, including Oculus, HTC Vive, and more. To facilitate compatibility with diverse VR hardware, we integrated the XR Plugin Management and the OpenXR package into our project settings. Moving forward, we incorporated a program to identify our specific headset, the Oculus Quest Pro. To optimize its performance, we tweaked a few settings, such as the rendering mode. We also designated Oculus as our plugin provider, enabling us to upload apk files and continually test our game throughout the development process.

5.3.2 Hands of the Player

As we started from the ground up, our initial setup was quite straightforward. We implemented virtual hands that mirrored the player's controller movements, along with a camera viewpoint that moved in sync with the headset, thanks to location tracking from the XR interaction toolkit.

There are three scripts that define the behavior of the player's hands in the game.

The first script, Hand.cs, controls the animation and behavior of a hand in the Unity scene. It relies on an Animator component attached to the same game object. The script includes public variables

to adjust the animation speed and references to game objects representing the knuckles and pointer. It handles the grip and trigger values, allowing smooth transitions between different animation states. In the Start() function, the script initializes the Animator component and disables the initial state of the knuckles and pointer objects to prevent them from appearing until a button is pushed. The Update() function is called every frame and calls two other functions: AnimateHand() to animate the hand based on the grip and trigger values, and CheckGripAndTrigger() to activate or deactivate the knuckles and pointer objects based on the grip and trigger values. The script also provides SetGrip() and SetTrigger() methods that are called externally to set the target grip and trigger values for the hand.

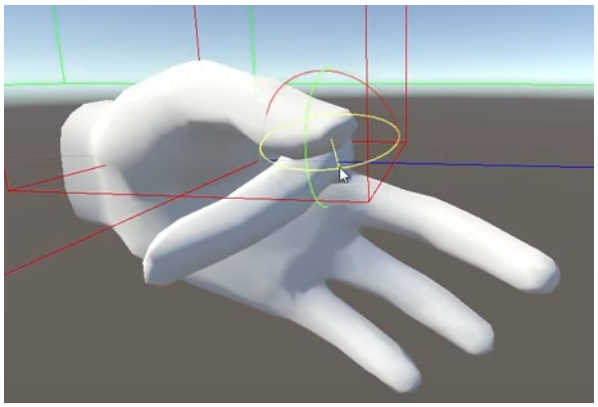

Figure 2: Process of animating hands for each finger

The second script, HandController.cs, acts as a bridge between an input controller and the Hand script. It requires an ActionBasedController component attached to the same game object. In the Start() function, the script retrieves the ActionBasedController component and assigns it to the controller variable. This allows us to define which controller (right or left) the individual hands resemble. The Update() function is called every frame and updates the grip and trigger values of the hand object. It retrieves the input values from the buttons of the controller and passes them to the corresponding methods in the hand object. By using these scripts together, the HandController enables the hand to respond to user input from an ActionBasedController and controls its animation and behavior by interacting with the Hand script.

The last script, ControllerVibration.cs, is specifically attached to the Knuckles-object, which is enabled by the Hand script when a combination of buttons is pressed. The knuckles are only visible and enabled when the hand is in the grip animation, and the pointer is only visible and enabled when the hand is in the trigger/pointer animation. This script handles the controller vibration when interacting with certain objects. It uses the XR input system to detect and control the vibration of a specific controller. The script relies on an XRNode to specify the input source (e.g., left hand or right hand). The red hand is set as the right hand, and the blue hand is set as the left hand. The script includes several public variables, such as the inputSource for specifying the XRNode and the interactableTag for assigning a tag name in the inspector. When an object with a matching tag enters the collider of the knuckles, it is immediately destroyed. Different tags are defined on each knuckle to destroy red crates with the red hand and blue crates with the blue hand. The TriggerHapticFeedback() method is used to trigger haptic feedback on the controller. It takes a duration and amplitude parameter. We used a high vibration intensity for a short duration to provide the player with physical feedback when hitting the corresponding crates.

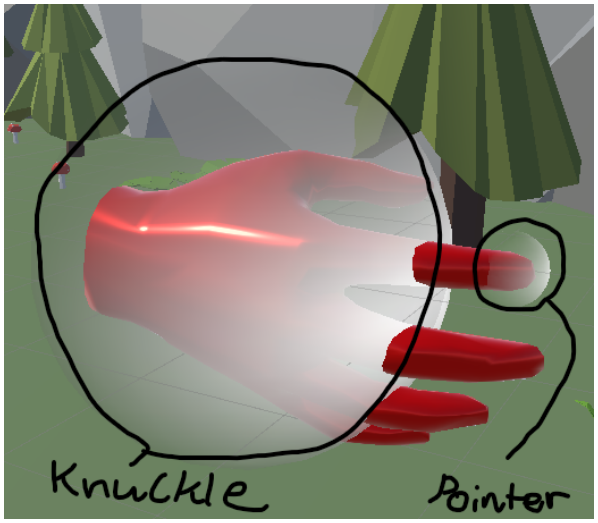

Figure 3: Knuckles and pointer

Additionally if the collider's tag matches the interactableTag, the script increments the combo count, plays a sound effect, and changes the speed using the SpeedManager. These additional features will be explained later. For now, it's important to understand the hand animations and how they enable the knuckles when the player "close the hand", which allows for detecting and destroying interactable objects.

5.3.3 Players position

The players position is simple and straight forward. We placed the player in the middle of the scene, and decided to keep the player in this static position, so we could easily implement game logic, without complication of calculating too much. This design choice allows us to move objects towards the player, simply by manipulating an objects Z-position. In the picture below, the red arrows show the process of subtracting the z-position value of objects like terrain and crates, making them move closer to the cart.

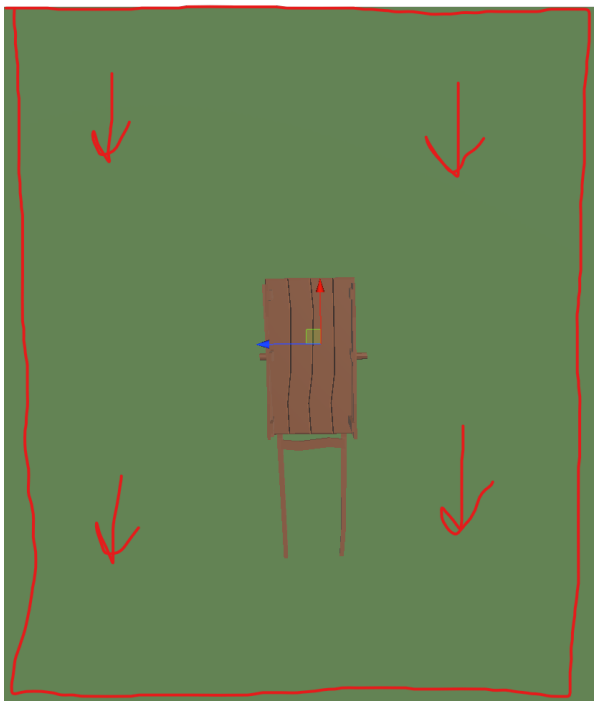

Figure 4: Demonstration of a hypothetical terrain moving towards the player

5.4 The Environment

The "environment" in a virtual reality context refers to all non-interactive elements that compose the player's visual landscape. It serves as the setting where the player finds themselves, including all observable elements except for those designed for direct interaction. The environment fundamentally shapes the player's visual perception of the virtual reality world.

Our selected aesthetic for the environment is a low-poly graphic style. This style is minimalist yet visually appealing and the Unity Asset Store offers a lot of readily available, free assets, simplifying our design process.

Low-poly assets typically feature a minimal number of textures, which greatly facilitates game rendering on lower-end devices. This characteristic makes them an ideal fit for hardware like the Oculus Quest Pro headset, which appreciates resource-efficient graphics for optimal performance.

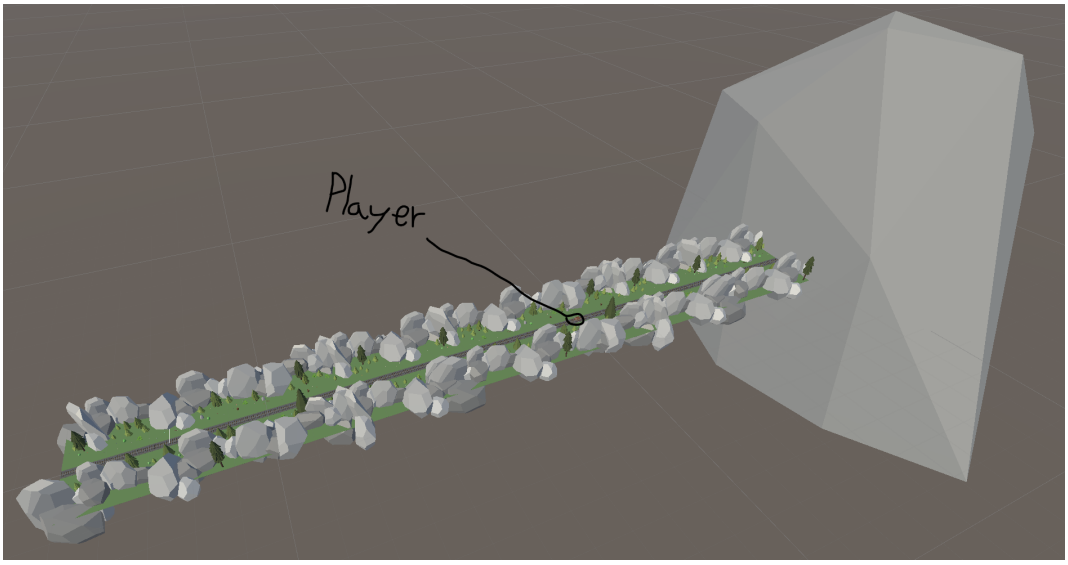

Figure 5: A zoomed out view of the environment, with the player outlined for size comparison

5.4.1 The Terrain

In our terrain design process, we started by handcrafting a terrain object, with the imported low-poly assets. We utilize this single Terrain object, replicated three times to form the entirety of our environment. We've crafted this Terrain object to ensure that when it's duplicated and placed sequentially, the transition between each copy appears smooth and continuous to the player. Special attention had to been given to the beginning and ending sections of the terrain, as these are the areas that align when repeated, effectively concealing the fact that it's the same terrain object being used, at least for some time. Although this approach is somewhat plain, it allowed us to focus our efforts on more complex tasks, such as level logic. It does however enhances the game's overall aesthetic and player experience.

Additionally, we've incorporated a visual feature to improve immersion. As the player's vision is directed towards a large mountain, it seems as if they are steadily approaching it, even though their position within the virtual landscape remains stationary. This illusion enhances the perception of depth and movement within the environment, by making the mountain look further away when the terrain objects are moving towards the player.

5.4.2 Terrain Mover

The process detailing how objects, including the terrain object, are moved on a per-frame basis will be covered in the later section titled 'Speed'.

However, there's a distinct aspect of how the terrain objects behave compared to other objects. To prevent an infinite extension of the terrain and to avoid an unnecessary increase in computational load over time, the terrain objects are programmed to teleport to a certain position, once a specific point z-position is reached. This ensures the game remains efficient, avoiding the potential for endless terrain generation that would put too much strain on the Oculus Quest's resources.

The "teleportation" takes place through a script named TerrainMover. It includes a FixedUpdate function which assesses whether the z-coordinate has fallen to -150 or below. If this condition is met, the command transform.position = new Vector3(transform.position.x, transform.position.y, 150); is executed. This line of code effectively resets the z-coordinate to 150. It aligns with the endpoint of the last among the three sequential terrain objects present in our game. This way we don't have to spawn any further objects, or remove them. We simply reuse the terrain, by moving it.

5.4.3 Enhancing the visual experience

By default, Unity's rendering approach can result in sudden transitions from invisible to visible objects, which can be distracting for the gameplay experience. To counter this issue, we've implemented a fog

effect. This not only ensures a smoother transition as objects come into view, but it also contributes to the illusion of distance, making the mountain appear farther away than it really is. The image provided below gives a side-by-side comparison of the player's perspective with and without the fog effect enabled.

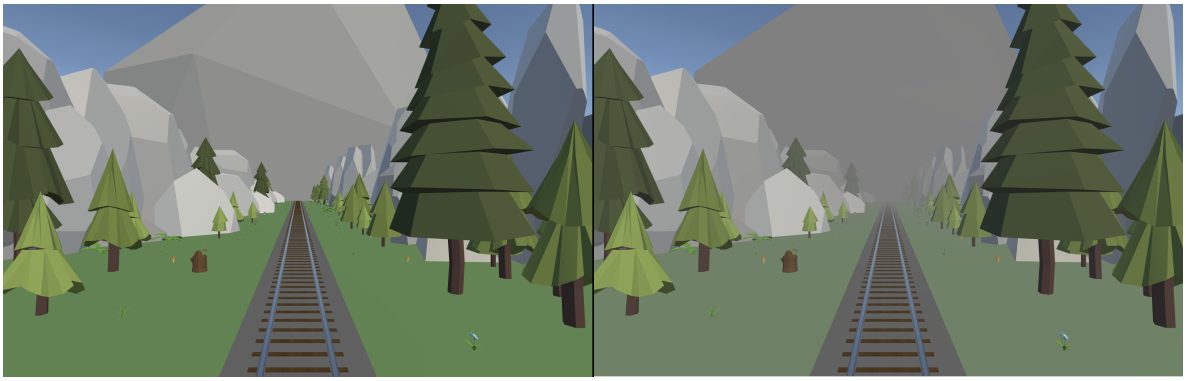

Figure 6: Scene without fog (left) versus with fog (right)

5.5 Interactive objects

In this section, we will explain the implementation of interactive objects in our game using the LevelManager, Level, SeriesManager and Series. You can also see the class diagram that visually illustrates their relationships and interactions here 5.2.

5.5.1 Series

A series in our project is composed of a list of series objects and a name. Each series object represents a game object and its corresponding position. The series and series objects are created as instances within the inspector, allowing for easy customization and configuration.

The SeriesObject class:

```
[SeriesObject class definition]
[System.Serializable]

public class SeriesObject
{
    public GameObject gameObject;
    public Position position;
}
```

As shown, the class contains two properties: "gameObject," which represents the game object/prefab associated with the series object, and "position," representing the position of the game object within the game world.

However, the Series class is defined as:

```
[System.Serializable]
public class Series
{
    public string name;
    public List<SeriesObject> seriesObjects;

    public Series()
    {
        seriesObjects = new List<SeriesObject>();
    }
```

```
}
```

This class consists of a "name" property to specify the name of the series and a list of "seriesObjects" to store the series objects belongs to the series. The constructor initializes the seriesObjects list to an empty list.

By using these classes, we can create and configure series objects within the inspector, allowing for easy management and customization of series in our game.

5.5.2 Series Manager

In short, the SeriesManager provides functionality for storing and retrieving the series, as well as maintaining a mapping between positions and corresponding vectors. We have implemented the SeriesManager class in to make it possible to create and manage th series within our game, through the inspector. The SeriesManager class acts as a central component responsible for handling series-related functionality.

Here is the representation of the SeriesManager code:

```
public class SeriesManager : MonoBehaviour
{
    public static SeriesManager Instance { get; private set; }

    public List<Series> seriesList = new List<Series>();

    public Dictionary<Position, Vector3> positionMap = new Dictionary<Position
        , Vector3>
    {
        { Position.RedSideUp, new Vector3(0.65f, 2.5f, 20f) },
        { Position.RedSideMid, new Vector3(0.65f, 2.0f, 20f) },
        { Position.RedSideDown, new Vector3(0.65f, 1.5f, 20f) },
        { Position.BlueSideUp, new Vector3(-0.65f, 2.5f, 20f) },
        { Position.BlueSideMid, new Vector3(-0.65f, 2.0f, 20f) },
        { Position.BlueSideDown, new Vector3(-0.65f, 1.5f, 20f) },
        { Position.Double, new Vector3(0.0f, 0.0f, 20f) }
    };

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
    }

    public Series GetSeriesByName(string name)
    {
        foreach (Series series in seriesList)
        {
            if (series.name == name)
            {
                return series;
            }
        }
        return null;
    }
}
```

Because we are using the inspector in Unity, we can create and configure series objects directly within the editor interface. The seriesList field within the SeriesManager allows us to add, remove, and modify series objects, that provide us with a user-friendly way to customize the series data, without the need for manual code modifications. Additionally, the redBox and blueBox prefabs are used for the series objects.

The SeriesManager class also includes a dictionary called positionMap. This dictionary maps each Position value with a corresponding vector that represents the position of the series objects (redBox and blueBox). This feature makes it possible for us to easily retrieve and position series objects based on their predefined positions.

The implementation of the SeriesManager class significantly improves the project's flexibility and maintainability. With the inspector, we can effortlessly modify and expand the series content. This streamlined approach minimizes the need for extensive code modifications and empowers us to efficiently iterate and improve the series functionality of our project.

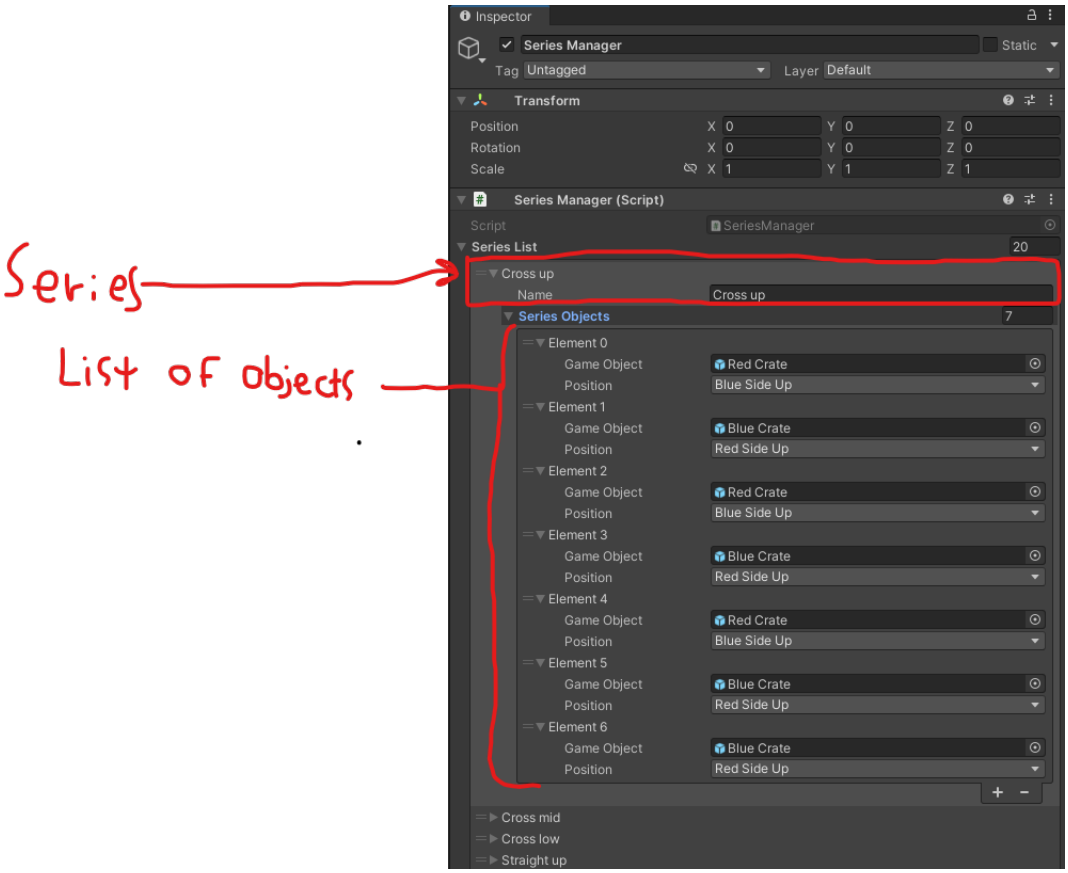

Figure 7: Series Manager in the inspector

In summary, the SeriesManager class facilitates the creation and management of series through the inspector. It streamlines the process of storing, retrieving, and customizing series while also providing a convenient position mapping feature. This improves flexibility and maintainability, making it easy to modify and expand the series content without extensive code adjustments.

5.5.3 Levels

In this section, we will describe what a level is and how it is managed. A level consists of a list of series. Each series represents a sequence of game objects.

The Start() method is responsible for initializing the level. It loops over each series name in the seriesNames list and retrieves the corresponding series using SeriesManager.Instance.GetSeriesByName(name). If a valid series is found, it is added to the seriesList.

```
    public List<string> seriesNames;
    private List<Series> seriesList = new List<Series>();
    private int currentSeries = 0;

    protected virtual void Start()
    {
        foreach (string name in seriesNames)
        {
            Series series = SeriesManager.Instance.GetSeriesByName(name);
            if (series != null)
            {
                seriesList.Add(series);
            }
        }

        ShuffleSeries();
        StartCoroutine(SpawnSeriesInOrder());
    }
```

After populating the seriesList, the ShuffleSeries() method is called to randomize the order of the series. The method uses the Fisher-Yates shuffle algorithm (also known as the Knuth shuffle) to ensure that every possible outcome/order is possible. This randomness adds variety to the game even if the levels and series remain the same.

```
//The Fisher-Yates Shuffle Algorithm
    private void ShuffleSeries()
    {
        for (int i = 0; i < seriesList.Count; i++)
        {
            Series temp = seriesList[i];
            int randomIndex = Random.Range(i, seriesList.Count);
            seriesList[i] = seriesList[randomIndex];
            seriesList[randomIndex] = temp;
        }
    }
```

Once the series list is shuffled, the SpawnSeriesInOrder() coroutine is started using StartCoroutine(). This coroutine spawns each game object in the series in a specific order. The objects are instantiated using Instantiate() at predetermined positions retrieved from SeriesManager.Instance.positionMap. A delay of 1 second is introduced between each object spawn using yield return new WaitForSeconds(1f). After all the series objects have been spawned, the level is considered complete, and LevelManager.Instance.LevelComplete() is called to proceed to the next level if available.

```
private IEnumerator SpawnSeriesInOrder()
    {
        foreach (Series series in seriesList)
        {
            foreach (SeriesObject seriesObject in series.seriesObjects)
            {
                Instantiate(seriesObject.gameObject, SeriesManager.Instance.
                    positionMap[seriesObject.position], Quaternion.identity);
                yield return new WaitForSeconds(1f);
            }
        }
        // Level complete
        LevelManager.Instance.LevelComplete();
    }
```

5.5.4 Level Manager

In this section, we will describe the logic behind object spawning in the game.

The LevelManager holds the preassigned levels for the game. Initially, it checks if another LevelManager exists in the game. If so, we delete this object because we should have only one manager. This practice, known as Singleton, prevents confusion regarding which manager the game should rely on. The deletion occurs within the Awake() method, called just before the game starts.

```
    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        else
        {
            Destroy(gameObject);
        }
    }
```

Moving on, in the Start() method, we verify if there are additional levels to run. If there are, we proceed by invoking the StartLevel() method.

```
    private void StartLevel(int levelIndex)
    {
        if (levelIndex >= 0 && levelIndex < levels.Count)
        {
            Instantiate(levels[levelIndex], transform);
        }
    }
```

The StartLevel() method checks if the index is bigger or equal to 0, and if it is smaller than the total count of levels in the list. If this condition is met, we instantiate the current level, corresponding to the index.

Upon level completion, the LevelComplete() method is called. This triggers the start of a coroutine function, which allows code to be executed gradually or with delays, without obstructing the main game loop. Within this coroutine function, we invoke WaitAndStartNextLevel(2f), causing a 2-second delay. After the delay, the level index is incremented to the next index in the list. Subsequently, we check if the index is still within the bounds of the levels count. If it is, we run the StartLevel() method with the current level index. Otherwise, a message is logged indicating that all levels have been completed.

```
 private IEnumerator WaitAndStartNextLevel(float delay)
    {
        yield return new WaitForSeconds(delay);

        currentLevel++;
        if (currentLevel < levels.Count)
        {
            StartLevel(currentLevel);
        }
        else
        {
            Debug.Log("All levels complete!");
        }
    }
```

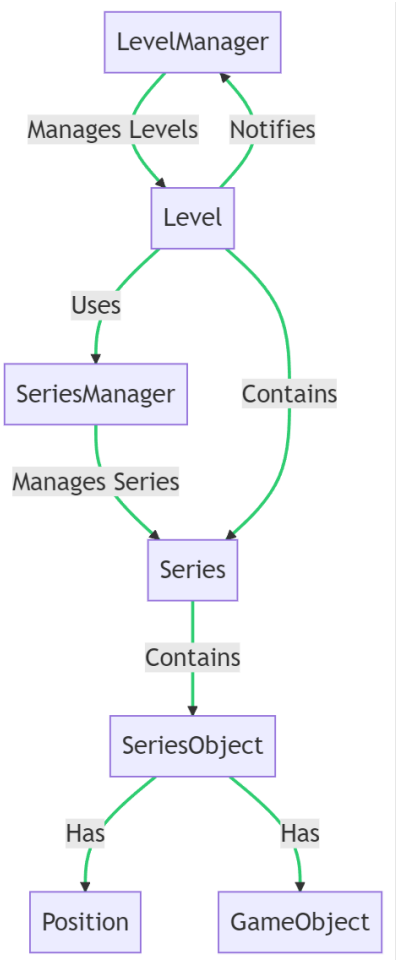

Figure 8: Illustration of how series and levels are linked and managed

5.6 Combo System

After successfully implementing the interaction between hands and hitable objects, we aimed to incorporate a basic combo system. This system would allow us to monitor the number of hitable objects that the player successfully hit in real-time, without any misses. To achieve this, we developed a straightforward approach that comprised of three fundamental concepts: the Combo Manager, the Combo Breaker, and the Combo UI.

5.6.1 Combo Manager

The Combo Manager's job is to keep track of the combo count in a game. This count is a number that starts at zero and can go up by 1 or be reset to zero.

The script has two public functions which other classes can access easily. We use a function called IncrementCombo() to add 1 to the combo count integer. The incrementCombo function is called once, the player hits the correct colored box, with the corresponding hand. We talked about the ControllerVibration script earlier. This script is connected to the player's hand controllers. When the KnucklesObject comes into contact with another object's collider, we use an if-statement to see if the other object has the interactableTag (Redbox for right hand or Bluebox for left hand). If the if-statement is true, we call the IncrementCombo() function. This way, we simply notify the ComboManager, when the player achieved to hit a box the right way. Currently we don't take in any arguments, like an integer, for the function. So IncrementCombo() only increases the combo by 1.

5.6.2 Combo UI

The ComboUI stands for Combo User Interface, and it shows the player their current combo. We use a variable called TextMesh, which is a component built into Unity's UI system. By using a TextMesh we can visualize text for the player to see. The ComboText script starts by rendering the text: "Combo: 0" and then we use an update function to update the text to "Combo: " +comboManager.combo.ToString(); By converting the integer "combo" to a string, we can add it to the string of the TextMesh. Finally we render the mesh by using the transform.localScale = new Vector3(size, size, size); Which refers to the transport component of the TextMesh. Note that the variable "size" is a float set to 0.1f. It's important to keep this in line in the update function, to ensure that the visual representation of the combo constantly is updated throughout the progression of the game.

Figure 9: The Combo Text in game (from a close perspective)

When playing the game this text will appear slightly below the players point of view, to avoid blocking vision.

5.6.3 Combo Breaker

The purpose of the combo breaker is, as the name suggests, to break the combo. In other words set the value of the integer "combo" to 0. This is done by using an invisible wall placed just behind the player. It's not visible to the player, but contains a collider, which interacts with all the boxes the player didn't manage to hit.

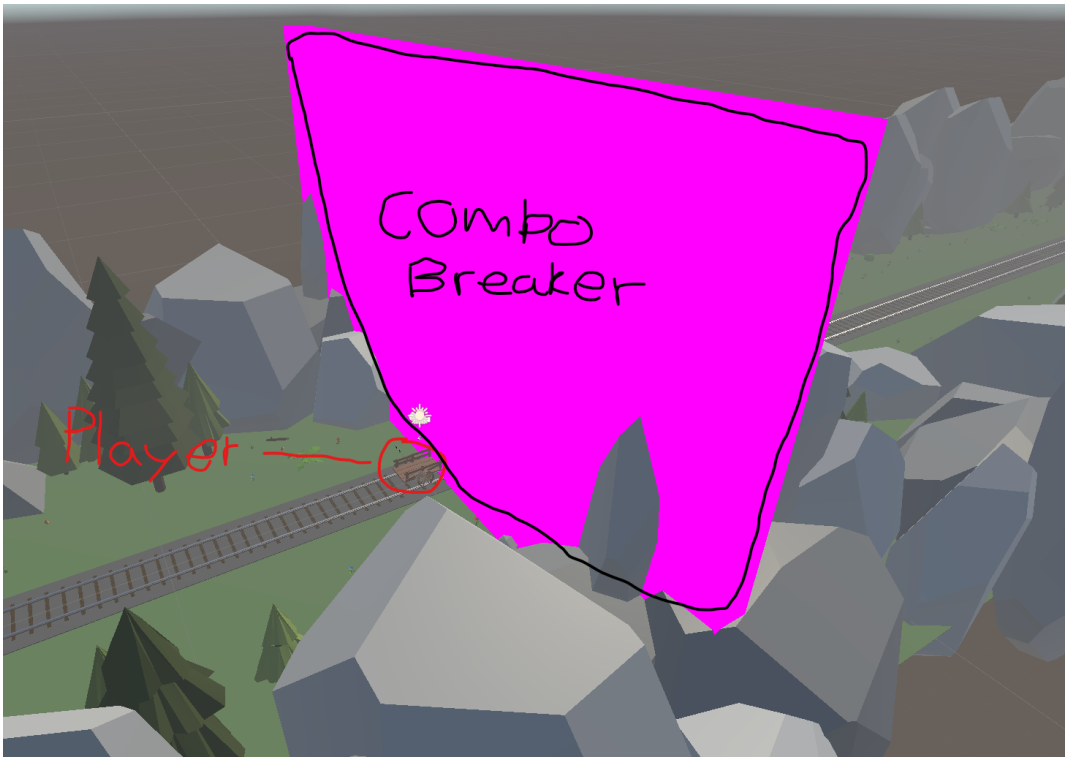

Figure 10: ComboBreaker in the scene, behind the player cart

We achieve this by calling the ResetCombo() in the script, attached to the ComboBreaker-Wall. The script is called crateDespawn, since the script originally only destroyed the crates that entered this space. It uses an if statement to check the tag for the boxes, just like the ControllerVibration script. If the object that enters, the ComboBreaker has the tag "RedBox" or "BlueBox" we notify the comboManager using this line: comboManager.ResetCombo();

However the comboBreaker also notifies two other managers, namely the SpeedManager and the SoundManager. Which will be explained in the following two sections.

5.7 Speed

In the design phase, we decided to make objects move towards the player instead of the player moving. This choice made handling objects much easier.

All objects in the Unity scene have a transform component, which allowed us to simply adjust one axis - the z-axis - to make objects move towards the player, given the player looks towards the z-axis direction. However, this method presented a challenge. It required all objects to be moved using a script. With a large number of objects like trees, rocks, flowers, and rails, this could potentially consume a lot of computational power. Therefore, we needed to find a solution to prevent this from becoming an issue.

It's possible to make objects children of other objects, and then move the parent object. This action causes the children to move along with it. The following image illustrates how children are position under parents.

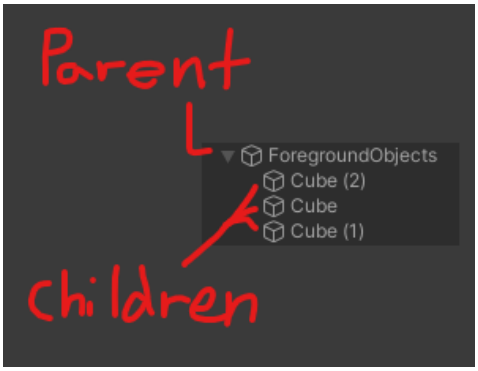

Figure 11: ForegroundObjects is the parent, Cubes are children

Consider the ForegroundObjects, which could be a single parent-object only containing a transform component. This can have a script attached to it to enable movement. Any child-objects linked to it would then follow its transform position, moving with it. This is a smart solution, but imagine a scenario where there are hundreds of child objects associated with this single parent.This was the case for our environment.

The transform component is just one of many components. Despite the shared movement, each child object still has its own transform, which is relative to the parent, and they also have other components. These include mesh filters and renderers that handle elements like lighting and shadows. So, while the movement is coordinated, the individual components of each child object still require processing.

To improve our game's performance, we imported the tool ProBuilder. This tool offers a feature called "Merge Objects," which allows us to combine multiple objects into a single entity. This merging process effectively removes the individual properties of the many objects, making further individual manipulation of their components impossible. Despite the drawback of not being able to manipulate all the hundreds of children, the process can significantly boost the scene's performance. When Unity renders a frame, it attempts to group as many similar objects as possible into a single batch. This batching process minimizes the number of commands the CPU must send to the GPU, thereby reducing the CPU's load and enhancing the game's performance. However, objects with differing properties, such as unique materials or textures, cannot be batched together. This results in the CPU needing to send more commands to the GPU, potentially slowing down the game.

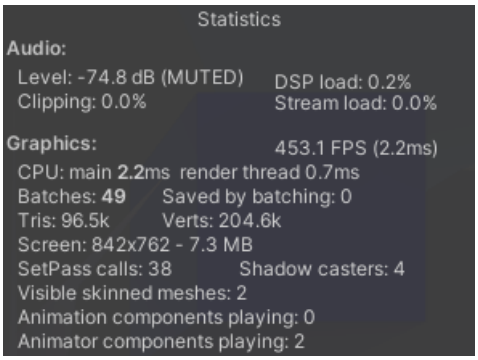

Figure 12: Measuring batches and FPS from the "Stats" window

By employing ProBuilder, we were able to drastically cut down the number of batches in the game. This reduction was a crucial requirement for our game to run smoothly on the Oculus Quest Pro.

The picture below shows the before and after result when using Merge Objects.

Figure 13: Before and after merging children

When we were satisfied with how the environment looked we merged it into single objects. Then we proceeded to add movement to them.

5.7.1 Speed of Objects

SpeedOfObjects is the script we use to move all the objects in the game, except the players hands and head, of course. The script only contains one function, the FixedUpdate(). This is a built-in unity function, similar to the Update() function, but what makes it unique is the fact that it's called on a fixed interval, which makes it ideal for physics calculations because it provides a consistent frame rate. If we used Update() the rate of the speed would be different on the headset, compared to the PC's we developed the game on. The Update() can also cause the speed to vary, whilst the FixedUpdate() avoids this issue.

In the FixedUpdate() of the script, a new position for the object is calculated. The new position is based on the object's current position (transform.position), but with an adjustment made on the z-axis. The adjustment is calculated by multiplying the speed of the object by the fixed amount of time that has passed since the last frame (time.fixedDeltaTime) The result is then negated to make the object move in the negative z-direction, or in other words towards the player, assuming the player looks straight ahead.

5.7.2 Speed Manager

The SpeedManager is implemented as a sort of "hivemind" since we change the speed variable according to how the player beforms. The picture below shows a visual representation of how what the SpeedManager does.

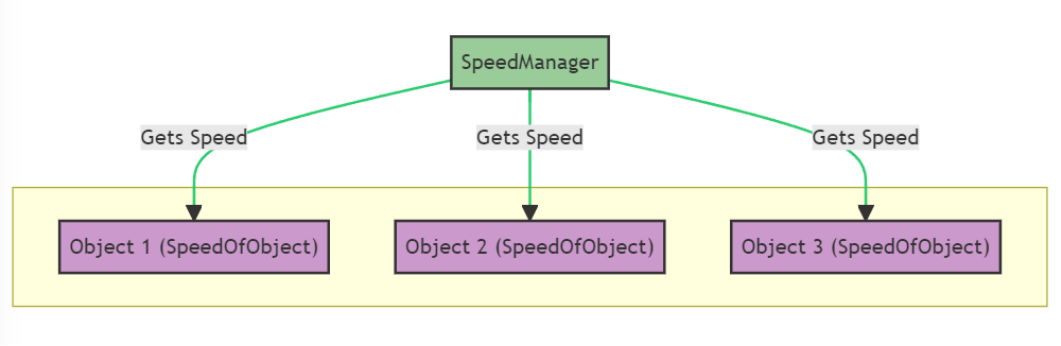

Figure 14: Speed is changed in the SpeedManager, SpeedOfObject retrieves the speed value from the SpeedManager

The SpeedOfObjects script retrieves the speed from the SpeedManager. This approach is efficient because it avoids the need to loop through all objects and individually change their speeds. Instead, all objects can reference the single speed variable in SpeedManager.

The SpeedManager script is a singleton class in Unity that manages the speed of objects in the game. By using a singleton pattern, we ensure that there's only one instance of SpeedManager throughout the game, which can be accessed globally. This makes it efficient as we only need to manage a single speed variable that can be changed frequently.

In SpeedManager, we have two float variables: minSpeed and speed. minSpeed sets a lower limit for the speed, while speed represents the current speed of the objects.

The ChangeSpeed function is a public method that allows other classes to modify the speed. It takes a float argument, speedChange, and adds this value to the current speed. However, if the speed is less than 3.5f, it resets the speed to minSpeed.

This function is invoked from two other classes.

The speed increases when the player hits a correct box, which is handled by the ControllerVibration script in the OnTriggerEnter function. The line SpeedManager.Instance.ChangeSpeed(0.1f); is used to increase the speed. Here, we're accessing the singleton instance of SpeedManager and calling the ChangeSpeed method with an argument of 0.1f, which increases the speed by 0.1f.

In a similar fashion, the speed is decreased when the player misses a box. This is handled by the ComboBreaker-Walls CrateDespawn script in its OnTriggerEnter. Here we use a negative float (-0.5f) to decrease the speed by 0.5f.

5.8 Audio

The implementation of audio in our game, is a feature that greatly impacts the player's experience. It serves as an immediate form of feedback for players, indicating whether they have successfully interacted with objects in the game or missed them.

Audio guides players in understanding the ongoing activities within the game, essentially playing a role in navigation. Before we implemented the feature of controller vibrations, audio was the primary source of feedback, and it continues to hold its relevance after implementing the haptic feedback. We also incorporated a high-tempo song to set an action-ready tone for the players.

To generate audio within Unity, two components are utilized, with the primary ones being Audio Sources and Audio Listeners.

Further in this discussion, we will illustrate how these components have been incorporated into our scripts.

5.8.1 Audio Source

An Audio Source in Unity is a component that can play an audio clip in the 3D environment. In our game it includes sound effects and music. Audio Sources act as the position of sound. In other words, they are the speakers in the 3D game world. It's possible to manipulate various properties of an Audio Source like controlling the volume and pitch. It's common practice to use many audio sources in unity,

especially in VR, especially if different objects have different sounds. The use of sources makes it possible to simulate several sounds from all kinds of different angles.

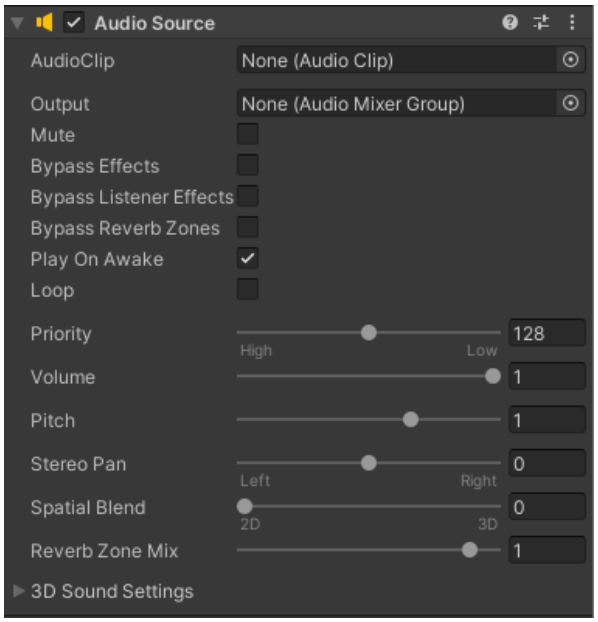

Figure 15: The Audio Source component seen from the inspector

As seen in the picture, an Audio Source requires an AudioClip for the playback of sound. However, in our setup, as visible in the project's snapshot, we didn't assign any default AudioClip to our Audio Sources. We have two distinct Audio Sources: one dedicated to music and the other to Sound Effects (SFX).

Rather than binding an AudioClip to each Audio Source directly, we went for a dynamic assignment approach through our SoundManager Script. This strategy grants us more flexibility and control over the sound playback, allowing us to alter or trigger different sounds as per the game state and player interactions.

We'll elaborate on the specifics of this SoundManager Script and its role in handling audio in the subsequent sections. Before exploring the script, it's important to understand the role of the Audio Listener.

5.8.2 Audio Listener

In the context of Unity's audio system, if we think of the Audio Sources as speakers — broadcasting sound into the environment — then Audio Listeners function as the player's ears, receiving the sound. The Audio listener is a simple component in terms of implementation. It takes position and direction into consideration when receiving audio from the Audio Sources, but most of audio manipulation lies in the Audio Source. The picture below shows the position of our audio listener, placed on the main camera. This way receiving sound is always tracked by the players position. This is the default case in every unity 3D scene.

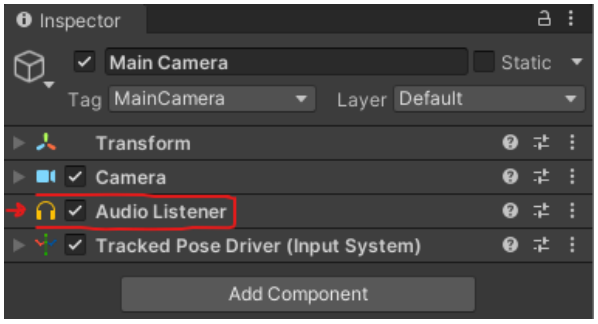

Figure 16: The Audio Listener as a component on the Main Camera

5.8.3 Sound Manager

As we mentioned earlier, the SoundManager script handles audio playback within our game. It uses two types of audio: Sound effects (SFX) and background music. We utilize two seperate AudioSource components for each type. If we relied on a single AudioSource for both background music and sound effects in our game, we would encounter a significant constraint: an AudioSource can only handle one AudioClip at a time. Therefore, any time a sound effect needed to be played, it would disrupt the continuous playback of the background music.

AudioSource sfxSource and AudioSource musicSource are the first two variables. They represent our sound effect and music Audio Sources, respectively. We assign these variables in the Unity inspector by dragging the corresponding Audio Sources into the script's serialized fields.

The third variable, AudioClip backgroundMusic, represents the audio clip for our background music. This clip is also assigned in the Unity inspector.

The Start() function in our script sets musicSource.clip = backgroundMusic, essentially assigning our chosen music clip to the music Audio Source. Subsequently, musicSource.loop = true ensures that this background music continuously loops, providing a consistent audio backdrop for our game. Finally, musicSource.Play() triggers the music to start playing when the game launches.

The final two variables, float lowPitchRange and float highPitchRange, establish the range for our sound effect pitch manipulation. These pitch alterations ensure the sound effects exhibit variation, preventing any monotony that might otherwise arise from repetitive sounds.

The PlaySound function, which other scripts call, accepts an AudioClip as an argument. Inside this function, sfxSource.pitch = Random.Range(lowPitchRange, highPitchRange) randomizes the pitch of each sound effect played, ensuring audio variety throughout the game.

Our PlaySound function from the SoundManager script is called in two distinct scenarios to enhance the game's feedback to player actions:

Within the ControllerVibration script, we invoke PlaySound when the player successfully interacts with a correct box. We do this through the line soundManager.PlaySound(crateDestroySound);. Here, crateDestroySound is the specific AudioClip played, which audibly signals to the player that they've successfully hit the correct box.

Similarly, we call the PlaySound function from the crateDespawn script, which is attached to the ComboBreaker wall. This happens when a player misses a box. The line soundManager.PlaySound(crateMissedSound); plays the crateMissedSound AudioClip, providing an audible cue to the player that they've missed a box. This provides immediate feedback to the player about their performance, enhancing the game's responsiveness and player engagement.

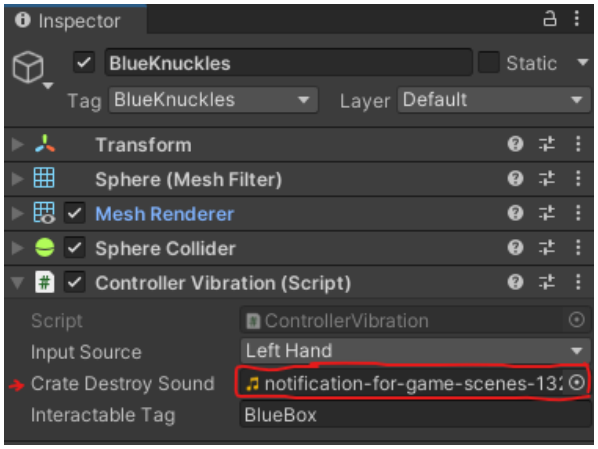

Figure 17: Example of a SoundClip being assigned

6 Evaluation (Correctness and Requirements)

Laves mandag, hjemme hos Niko, sammen.

7 Conclusion

7.1 Future Work

As the natural next step, we want to create a calibrator that measures the arm length and squat ability of the player before entering the game. Next, we want to create a UI/User feedback that gives the player their high score and current combo.

References

[1] U. Technologies, "Unity manual - components." https://docs.unity3d.com/Manual/Components.html [Accessed: April 29, 2023], 2021.

[2] U. Technologies, "Unity script reference - rigidbody." https://docs.unity3d.com/ScriptReference/Rigidbody.html [Accessed: April 29, 2023], 2021.

[3] U. Technologies, "Unity script reference - transform." https://docs.unity3d.com/ScriptReference/Transform.html [Accessed: May 31, 2023], 2021.

[4] U. Technologies, "Unity script reference - collider." https://docs.unity3d.com/ScriptReference/Collider.html [Accessed: April 29, 2023], 2021.

[5] U. Technologies, "Unity script reference - material." https://docs.unity3d.com/ScriptReference/Material.html [Accessed: May 22, 2023], 2021.

[6] U. Technologies, "Unity manual - prefabs." https://docs.unity3d.com/Manual/Prefabs.html [Accessed: April 29, 2023]https://docs.unity3d.com/Manual/NestedPrefabs.html, 2021.

[7] U. Technologies, "Unity manual - nested prefabs." https://docs.unity3d.com/Manual/NestedPrefabs.html [Accessed: April 29, 2023], 2021.

[8] U. Technologies, "Unity asset store." https://assetstore.unity.com/ [Accessed: May 22, 2023], 2023.